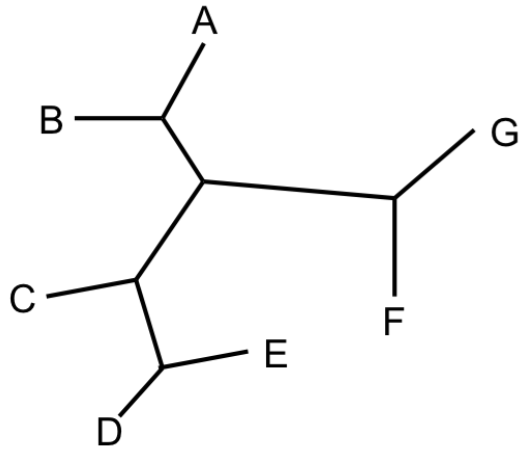
PROBLEM SET 8

Due Friday, March 6th, by 3:00pm through Canvas. Assignments turned in more than 5 minutes late will be penalized 10 points, with an additional 10 points every 24 hours thereafter.

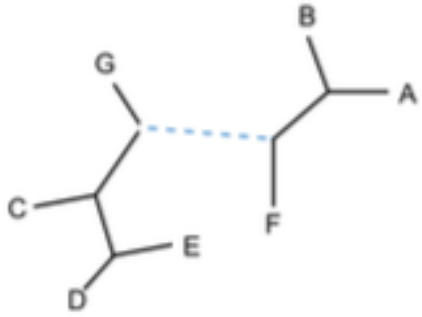
PART A. ALGORITHMS (40 POINTS)

1. (20 points)
1a. (10 points) Recall the hill climbing algorithm we described in class. Such an algorithm is often referred to as a "local" search. Explain, in your own words, what's local about this search.

1b. (10 points) Draw all the possible trees resulting from Nearest-Neighbor Interchanges of the following tree. Hint: first circle all **internal** branches and figure out how many trees you expect to end up with based on the logic from class to exchange branches. For each tree, indicate the internal branch around which you exchanged branches (example below).



Example of one tree with branch over which exchange is made indicated by dotted line:



2. (20 points) You have been given a (tiny) RNA-seq dataset, indicating the number of times transcripts from four genes were observed in 3 cells. You think this data represents two cell types and you would like to distribute these cells among two clusters.

	gene 1	gene 2	gene 3	gene 4
cell 1	3	6	9	1
cell 2	4	2	5	3
cell 3	9	1	4	2

2a. (10 points) Calculate the Euclidean distance between each pair of cells. You do not need to write a program to do this (but at this point you may find it easier to do so than doing this by hand – isn't that neat?), but please show your work. If you're not sure how to calculate Euclidean distance, here is a good resource: http://rosalind.info/glossary/euclidean-distance/

2b. (10 points) Based on these distances, how would you split these cells across two clusters?

PART B. PROGRAMMING (60 POINTS)

3. (60 points) In class, we learned about the hill climbing algorithm, which is useful for finding peaks (or valleys) in datasets where it is not possible to sample all positions and look for the highest (or lowest) one. In this problem we will implement a hill climbing algorithm to run on a one dimensional representation of peaks. For this problem, you will turn in a single program called hill_climbing.py, as well as some written responses.

3a. (20 points) small_landscape.txt is a file containing a string of numbers representing a few peaks and valleys. The values represent the y-coordinates ("heights") while the position in the string represents the x coordinate. (e.g. [1,1,2,3,4,3,1,0,-1,2,3] represents a hill with a peak at 4 followed by a valley with a low point at -1). Write a **function** called locate_peak() which takes in a list of values like the one above and the number of steps to take (n), implements the hill climbing algorithm, and returns the peak it reaches after n steps.

Your function logic should be something like this:
Choose a random starting position in your list
For each of n steps:
    Randomly choose whether to move right or left
    Move if new value is >= old value; don't move if new_value < old_value.
    - (make sure you are moving when the values are equal – otherwise you'll get stuck.)
    - (also make sure you are not falling off the ends of the list).
Return the final value ("height") after n steps

Feel free to use randint() from the random package.

Generate a list from small_landscape.txt (as an input argument) within your program and test your function with different values of n to see how your result changes. Run your function with the same value n multiple times and think about the results. No need to turn anything in yet.

3b. (10 points) You may have noticed that your function returns different values over multiple runs with the same input (let's say 10). Describe in words (a) what this mean about topology of your landscape; (b) how the size of n impacts how likely you are to reach the true peak; and (c) whether you think you are guaranteed to reach the true peak with a large enough n.

3c. (5 points) Hopefully, you noticed a relationship between the number of steps taken during hill climbing and how variable your final height is with each run of the function. Let's quantify this. Add some code to your program where you define the number of steps (n) to take, run your algorithm 100 times with that number of steps, store the output heights in a list, and use the set() function to count the number of unique heights reached during the 100 iteration. Have your program print n and the number of unique heights in the form:
10 : 26
(where n = 10, and 26 is the number of unique heights reached in 100 iterations. This will vary a bit every time you run the program, but for n = 10, mine is close to 26 each time.)
What is the number of unique heights reached at n = 10, n = 100, and n = 1000?

3d. (10 points) Now let's make the program systematically count the number of unique heights over 100 iterations for a variety of n values. First, add some code to generate a list of n values which represent 2^x up to (and including) 1024. Your list should look something like this: [2,4,8,16,…,512, 1024]. Python's "squared" notation is ** (2**3 = 8). A while loop might be helpful here!!
Then, modify your program to run the hill climbing algorithm 100 times for each value of n and save the number of unique peaks reached associated with that value n (same as the output from 3c.) Finally, print the results in the format below. Turn in this program, which should include your function.

INPUT:
> python hill_climbing.py small_landscape.txt

OUTPUT:
2 : ##
4 : ##
8 : ##
…
1024 : ##

3e. (5 points) Based on your output from 3d, how many peaks do you think exist in this dataset?

3f. (10 points) Run your code on a new input file, large_landscape.txt, and paste the output here. Is the number of peaks in your data obvious from this output? If not (and it shouldn't be ☺), give two possible explanations. Hint: one of these explanations can be addressed by modifying the code, and one cannot.